Hydraulics Research
Wallingford

USING MICROCOMPUTER GRAPHICS TO DISPLAY NUMERICAL MODEL RESULTS

Stuart Davidson

Report No. SR 52
May 1985

HYDRAULICS RESEARCH STATION
WALLINGFORD, OXON.
- 6 JUN 1985
CLASS No. .....
ACC No. 85/6/42

Registered Office: Hydraulics Research Limited,
Wallingford, Oxfordshire OX10 8BA.
Telephone: 0491 35381. Telex: 848552

©Crown Copyright 1985

Published by permission of the Controller of Her Majesty's Stationery Office.

This report describes work carried out under Contract DGR/465/31, funded by the Department of Transport from April 1982 to March 1984 and thereafter by the Department of the Environment. Any opinions expressed in this report are not necessarily those of the funding Departments. The DoE (ESPU) nominated officer was Mr A J M Harrison. The work was carried out by Mr S Davidson in the Computer Services Department of Hydraulics Research, Wallingford. The contract was managed by Dr A J Brewer. This report is published with the permission of the Department of the Environment.

CONTENTS

1 INTRODUCTION

Early in the investigations into the feasibility of using a micro-computer based colour graphics system at Hydraulics Research showed that writing programs to efficiently control such a system requires an understanding of raster techniques and colour theory.

In order to obviate the necessity for the Hydraulic engineer to acquire such an understanding, research was aimed at designing programs that allowed them access to the colour graphics facilities through easy-to-use cursor and keyboard controls.

To this end, the following programs were designed and written:

(a) An editor to manipulate displayed images.
(b) A print program for displaying hard copies of colour images.
(c) A program for displaying result files from mathematical models as an animation sequence.

Methods of video recording or filming the display images were discussed with the Photographic Section and advice was sort from outside companies. Several were tried and investigations continue.

As the research progressed, further areas of interest presented themselves and the shortcomings of the basic kit being used became apparent.

The 'eye catching' feature of colour images could be usefully exploited for publicity purposes. A graphical presentation of Hydraulics work would be visually striking and informative.

To produce more meaningful realistic images a wider range of colours would be required. If this were combined with a device such as a pallette or light pen that exploits the user-friendliness of graphics then the system would be open to a wider range of design applications at Hydraulics.

Animation of results from mathematical models helped to clarify the results. To extend this facility to more complex models, a more powerful processor would be required. To satisfy this requirement a sophisticated graphics terminal could be connected to the mainframe computer. This would also circumvent any restrictions imposed by transferring results to the micro-computer.

Accuracy would be improved with a higher resolution, this would enable more complex images to be presented.

Some of these improvements could be achieved at little extra cost.

2 DESCRIPTION OF GRAPHICS EQUIPMENT

2.1 The graphics controller

Pluto is an intelligent graphics controller. In its simple form the system resides on a single board which contains the following:

Intel 16-bit micro-processor
192 K bytes Frame buffer
Graphics routines in firmware
Expansion Bus for adding extra features

The version of Pluto used for this investigation was the basic board with a high resolution option. This provided a choice of eight colours and a resolution of 640H x 288 V or 576V in the high resolution mode.

The system was connected to an ACT Sirius 1 micro computer which also uses the 8088 micro-processor. With the Pluto card housed in an external case, connection to the Sirius was through an interface card in the Sirius expansion BUS.

To display images, a Cotron Sword monitor was connected to the Pluto.

To access the graphic routines from a high level language such as Fortran, an assembler library is required.

The purpose of the routines in the library is to act as an interface. Sending data from the calling program to the address in the hosts memory to which Pluto in connected.

The assembler library originally supplied with the system was not suitable for Microsoft Fortran as used on the Sirius. The reason for this was that this version of Fortran is able to address outside the 64K limit normally imposed by 16-bit micro-processors and so it uses 'long data pointers'. These are effectively 20-bit wide addresses composed of a segment address and an offset within the segment.

Another assembler library was ordered which was able to cope with the longer addressing space. This library did not provide an interface to all the Pluto graphics routines and so some of these had to be written.

2.2 The Frame buffer

On the basic Pluto system the frame buffer is 192K bytes in size. This relatively large amount of memory is required because Pluto uses raster techniques to display the image.

With raster displays, each picture element (pixel) on the screen is controlled by a memory location in the frame buffer.

The value of a location determines the colour of the corresponding pixel.

The image is displayed on the screen by scanning the frame buffer in a similar manner to that in which a TV screen is scanned. That is, horizontal lines from the top left corner down to bottom right corner and repeat.

The frame buffer must be scanned and the image refreshed onto the screen many times per second in order to achieve a 'flicker free' display.

In Pluto's low resolution mode, the image is completely refreshed onto the screen every 1/50th of a second.

To achieve the higher resolution of 640H x 576V, Pluto uses interlace techniques.

Interlace is a means of producing a high resolution image without the associated flicker problems of scanning a larger frame buffer. This is achieved by scanning only half of the frame buffer each time. On one pass the locations controlling even numbered pixel lines are scanned and on the next pass, locations controlling odd numbered pixel lines are scanned. The result is that new information is still being displayed every 1/50th of a second.

In the version of Pluto used, each location controlling a pixel consisted of 3-bits. This gave a range of eight possible colours.

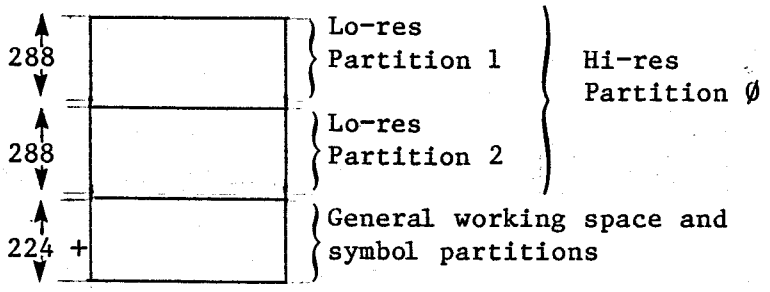
These 3-bits can be thought of as controlling the three primary colours red, green and blue. Therefore, the whole frame buffer can be thought of as three colour planes.

The various combinations of bits being set/unset produces the range of colours: Black, Green, Blue, Cyan, Red, Yellow, Magenta and White.

The 192K frame buffer supports 640H by more than 800v pixels:

(i.e. 640 x 800 x 3 = 1536000 bits = 192000 bytes)

The frame buffer is divided up in the following manner.



The frame buffer is dual ported enabling it to be updated and displayed on the screen simultaneously.

2.3 The colour printer

A Diablo Series C ink jet printer was connected to the parallel port on the Sirius.

The Diablo provides four basic colours: the subtractive primaries cyan, yellow and magenta. Black is also provided because, in printing, it is difficult to obtain a deep black by combining colours.

The colours cyan, magenta and yellow are the complements of the additive primaries red, green and blue respectively. These subtractive primaries are provided because in common with other colour hard copy devices the Diablo is based on the CMY colour model.

The basis of the CMY colour model is that colours are specified by what is removed from white light. White light is the sum of the additive primaries red, green and blue.

As an example: removing red from white light leaves green and blue and these combine to produce the subtractive primary cyan.

Therefore, a surface coated with cyan coloured ink absorbs red light and reflects blue, green. Also, a surface coated with cyan and yellow absorbs red and blue and reflects green from illuminating white light.

Whilst the Diablo is based on the CMY colour model, the Pluto is based on the RGB colour model.

The basis of the RGB colour model is that colours are derived from what is added to black rather than what is subtracted from white light.

The individual contributions of each primary are added together to form the resulting colour.

Knowledge of both the CMY and RGB colour models was important when writing a routine to copy a screen image from Pluto to the Diablo.

3 DESCRIPTION OF GRAPHICS SOFTWARE

3.1 The hard copy facility

This facility produces hard copies for screen images. The following requirements dictated that the routine should be written in a language other than Fortran:

- The routine should be as fast as possible
- Perform logical operations on all bits in bytes
- Send commands direct to the printer port (unformatted)

To satisfy these requirements, the routine was written in 8086 Assembler and incorporated in the assembler interface library.

The Diablo provides a bit pattern printing mode which prints four pixel lines at a time. The format of the command is: information specifying the colour, number and length of the pixel line followed by a string of bytes specifying the bit pattern to be printed.

Reading the Pluto bit planes into individual strings of bytes formed the basis of the bit pattern required in the printing command.

The following example of printing cyan on the Diablo demonstrates reading individual bit planes and producing a bit pattern for the print command.

Line of pixels	-	B l a c k	G r e e n	B l u e	C y a n	R e d	Y e l l o w	M a g e n t a	W h i t e
Read Red plane	-	0	0	0	0	1	1	1	1
Read Blue plane	-	0	0	1	1	0	0	1	1
Read Green plane	-	0	1	0	1	0	1	0	1

Perform logical OR on bytes holding blue and green planes:

```
  0 0 1 1 0 0 1 1
  0 1 0 1 0 1 0 1
  ---------------
  0 1 1 1 0 1 1 1
```

Perform logical NAND with byte holding red plane:

```
  0 1 1 1 0 0 0 0
```

This will remove any bits which represent pixels with red in them. This is effectively subtracting red from white light as detailed in CMY model example.

the bit pattern printing command and this byte would now be sent to the Diablo to print cyan.

By repeating this process and using the appropriate bit planes at each stage we also print magenta and yellow.

The result is that both the subtractive and additive primaries are produced on the Diablo. The additive primaries are produced where the subtractive primaries overlap. For instance, where cyan and yellow overlap green will be produced.

Note: the Diablo does not print blue because magenta and red overlap to produce violet, however, this is usually adequate.

Printing black required a slightly different approach to that used for the subtractive primaries. The reason for this was that colours in the RGB model combine to produce white whereas they combine to produce black in the CMY model (i.e. a location in the frame buffer with bits set in the red, green and blue planes would produce a white pixel whereas, printing cyan, magenta and yellow on the Diablo would produce murky black.)

To obtain a bit pattern representing black pixels perform a logical NOT on, say, the red plane:

1 1 1 1 0 0 0 0

Perform logical NAND with, say, the blue plane:

1 1 0 0 0 0 0 0

Perform logical NAND with the green plane:

1 0 0 0 0 0 0 0

The resulting bit pattern represents the only black pixel in the example. This would now be sent to the Diablo as part of the command to print black.

Coping with distortion in hard copies

To enable an image from one of the low resolution partitions to fill the screen, the refresh mechanism

leaves alternate pixel lines blank. These blank lines are only noticeable when viewing the screen from a very short distance.

The result is that taking a copy of all locations in the partition would produce a print such as Fig 1.

Whereas the ratio of width to height should be approximately 1:1 as displayed on the screen.

To improve the proportions of the picture blank lines could not have been introduced in the hard copy since these would have been noticeable. Instead, every pixel line was duplicated.

Essentially the screen dumping routine detects when a low resolution partition is being displayed. Normally the routine would read 4 pixel lines at a time since the Diablo prints 4 at a time. However, in the case of low resolution only 2 pixel lines were read. A 'bit shuffling' routine then expanded the string of bytes into a string representing 4 pixel lines.

Although the 'bit shuffling' routine had to be used on each bit-plane for every two pixel lines it was not a considerable time overhead. This was because it used hardware implemented string operators in the 8086 Assembler.

Using the 'bit shuffling' routine produced pictures such as Fig 2.

The resolution of the Diablo is 1024H x α and the Pluto is 640H x 288V (or 576V). Since the physical width of the paper and the screen are similar this means that a pixel on the Diablo is approximately 2/3 the size of a Pluto pixel.

The result is that a full screen dump of the Pluto appears 2/3 the size on the Diablo.

To overcome this deficiency it would be possible to restrict the amount of the screen to be dumped to 512v in high resolution mode or 256V in low resolution mode. Then double each pixel in both directions and print the image sideways on the Diablo. Note: in low resolution each pixel would be quadrupled vertically.

This option may be provided in future.

3.2 The result file display facility

The result files used for display purposes, during the investigation were produced on the ICL 2972 mainframe.

These were transferred to the Sirius by using a terminal emulation package on the micro. This

package makes the micro appear to the mainframe as a Remote Job Entry (RJE) station. The RJE station comprises an operating console, virtual line printer and two interactive terminals.

Listing a result file from any mainframe terminal to the micro's virtual line printer and controlling the transfer from the RJE console enabled the file to be placed on the micro's disk.

Format

The files represented regular grids with real numbers defining the value of each grid cell. These real numbers represented a wide range of values. However, since Pluto could only display eight colours and the codes for these were the integers 0-7 there was initially no need to transfer the file with real numbers.

A routine was written and used on the mainframe to convert ranges of real numbers into single integers. It was felt that the time saving in transferring values as single byte integers rather than, say, six byte reals would be valuable. Especially when transferring files containing large numbers of grids.

Displaying (cell for cell)

To handle result files on the Sirius a Fortran program was written.

The program read the file header which contained information on the dimensions of the grid and the number of grids in the file.

The grid dimensions were given in terms of the number of cells horizontally and vertically. If the number of grids was greater than one it was assumed that a sequence was to be shown and so the low resolution option was used. This is explained more fully in the animation section.

To determine the dimensions of a grid cell in pixels:

Number vertical pixels =

$$\text{Integer value of } \left(\frac{\text{Vertical resolution}}{\text{Number vertical cells}}\right)$$

Number horizontal pixels =

$$\text{Nearest integer value of } \left(\text{Number Vertical pixels} \times \text{Scaling factor}\right)$$

If:

((Number horizontal pixels x number vertical pixels) > horizontal resolution) then

Decrement number vertical pixels and re-calculate

With the dimensions of a grid cell determined in pixels, the program then read the values for a grid into an array. It then stepped through the array plotting the value of each grid cell as a coloured rectangle.

Using this method and with all grids held in memory, the time to update the screen was approximately 2 seconds. This is not only unacceptably slow but limits on the amount of available memory could also be quite restrictive.

Limits on memory

It has been mentioned that Microsoft Fortran can address outside the 64K code segment limit. The limit on a linked Fortran module is 384K.

This extra space can be used to store arrays if they are placed in named COMMON blocks since named COMMON blocks are automatically placed in separate segments. A segment may be up to 64K bytes in size which is also the maximum size of a Fortran array.

If a typical grid produced on the DAP represented a 64 x 64 array and each element or grid call was represented by a single integer in a byte then the grid would occupy 4K bytes. Therefore, sixteen such grids could be held in one array in a named COMMON block.

Up to five of these maximum sized COMMON blocks could be declared in the 384K link module since at least 64K will be required for the Fortran code and assembler interface library etc.

The result is that an animated sequence could contain approximately 80 such grids, all held in memory. This could not only prove restrictive but the time to read the file (s) into the arrays was unacceptably large.

To overcome the restrictions of this method of display and holding all grids in memory, Run Length Encoding was used with unformatted files.

Run Length Encoding (RLE)

In a typical image the amount of continuous space of the same colour is often quite large. The basis of RLE is both to reduce the amount of information required to code this space and reduce the amount of time to display it.

For instance, a line of 64 grid cells all the same colour would normally require 64 bytes of data to code them and 64 separate rectangle plotting commands to display them. However, with RLE the amount of data can be reduced to 2 bytes. One representing the colour and the other representing the number of cells. The program interpreting this RLE data would know the dimensions of a grid cell and so just one rectangle plot command would display the whole line of cells.

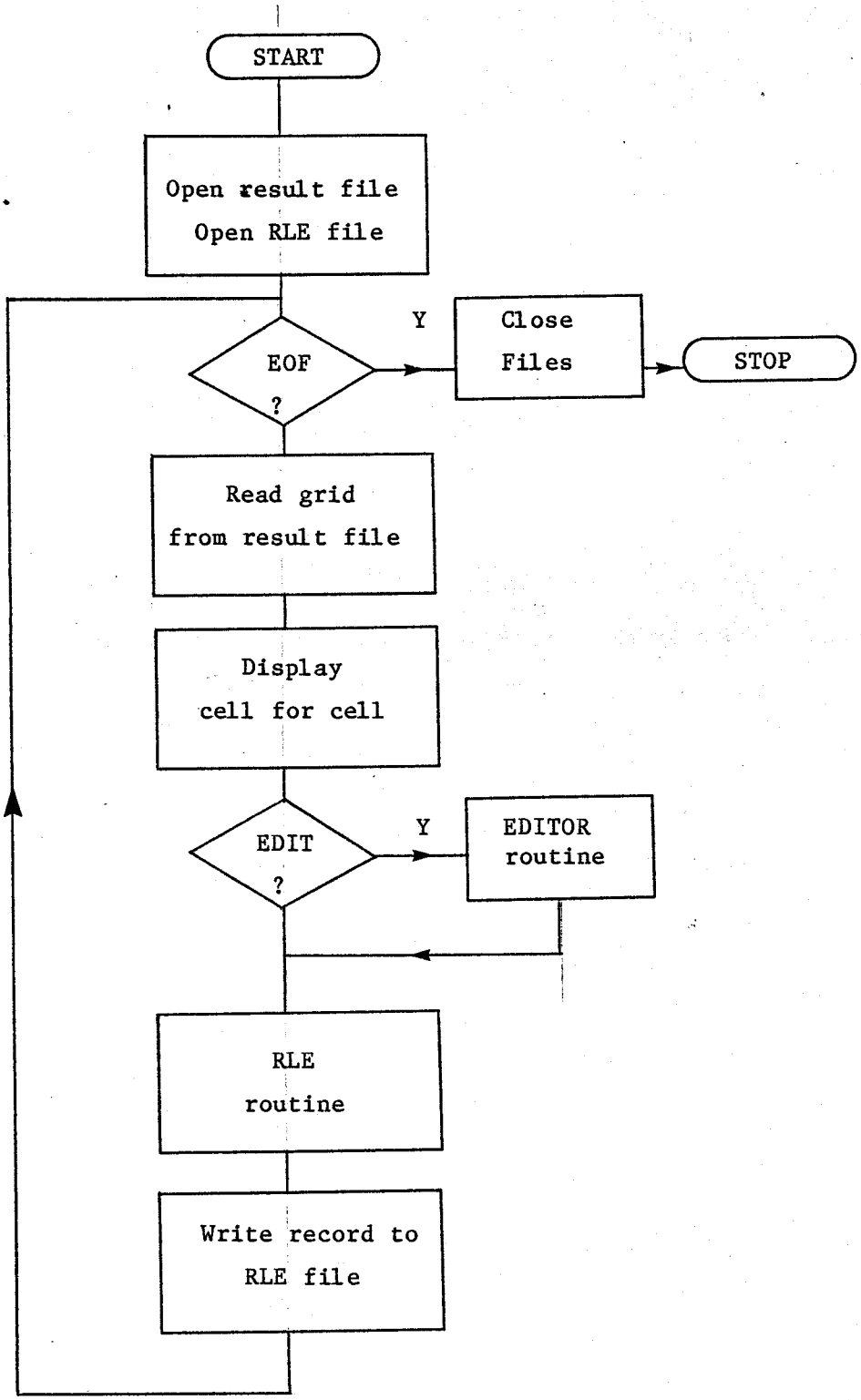
Unformatted files were used for storing the RLE data because these are more efficient, in terms of speed, than formatted. Sequential access was used since this is faster than direct access.

A disadvantage of RLE is that it can be difficult to edit the data. For instance, if the colour of a grid cell in a line of cells of the same colour is to be changed. Then a new field would have to be introduced, assuming a record represents the whole image.

For this reason RLE was not used on the mainframe before files were transmitted even though the possible data compression would have been greater than the real to integer conversion mentioned earlier.

Produce RLE data

The following flow chart demonstrates how results files from the mainframe were converted to files containing records which represented grids in RLE form.


START
Open result file
Open RLE file
EOF
?
Y
Close
Files
STOP
Read grid
from result file
Display
cell for cell
EDIT
?
Y
EDITOR
routine
RLE
routine
Write record to
RLE file

The RLE routine skipped to the top left corner of each cell as displayed and returned the value of the pixel. In this way an array was built up which included any changes made by the editor. The contents of this array were then run length encoded and written to the output file as a single record.

Producing animation

In an animation sequence the viewer should not see the screen being updated. This can be achieved on Pluto by using the two low resolution partitions in the frame buffer.

The program was written to display a sequence of grids. The program designated one partition to be updated and the other to be displayed. A record was read from the RLE file into an array, this run length encoded data was written to the partition to be updated. When complete, the assignment was swapped and the next record was read, etc. This reduced the update time to approximately 1 second. Also, limits on memory were not a problem because data for only one grid was held in memory at any one time.

Distortion of displayed images

Low resolution images are able to fill the screen by leaving alternate blank pixel lines, whereas high resolution images use all pixel lines.

This results in displayed images requiring different scaling factors to be applied, depending on whether a high or low resolution screen is used.

Applying these scaling factors would very likely result in lengths with a decimal fraction. When these are displayed in pixels they must be whole numbers since a pixel is the smallest displayable element. The conversion to integer inevitably introduces rounding errors. This is known as the 'nearest integer' method.

If this rounding error is present in the width of each grid cell, it is possible that a noticeable distortion would appear in a line of, say, 64 cells.

Applying the scaling factor to lengths of cells of the same colour, as would be possible with RLE data, prevents the cumulative error building up but the overall width of the image becomes unpredictable since the number of colour changes is not known. Therefore, this method is less desirable than the previous one.

In an effort to circumvent the problem of a cumulative error building up, the following method was used. It is best explained by pseudo-code.

```
     REAL-CELL-WIDTH     = HEIGHT_OF_CELL x
                           SCALE FACTOR
     SUM-OF-CELL-WIDTHS  = 0

FOR N = 1 TO NUMBER-OF-HORIZONTAL-CELLS

     SUM-OF-CELL-WIDTHS  = SUM-OF-CELL-WIDTHS +
                           REAL-CELL-WIDTH
     CELL-WIDTH(N)       = NEAREST INTEGER
                              (SUM-OF-CELL WIDTHS +
                              REAL-CELL-WIDTH) -
                           NEAREST INTEGER
                              (SUM-OF-CELL-WIDTHS)
     IF CELL-WIDTH (N)   = Ø THEN CELL-WIDTH (N) = 1
REPEAT
```

NB: A cell with width Ø is avoided because this would result in data being lost from the displayed image.

This method prevents the cumulative error exceeding ½ a pixel. However, when this is compensated for the effect is quite severe in that a pixel is dropped. This could cause problems when 'butting' multiple grids together in the same display.

Also, preventing a cell width of Ø effectively means that a fine grid of approximately 1 pixel per cell is able to build up a cumulative error. This would certainly cause problems when trying to 'butt' such a grid onto ones which have no cumulative error.

The result is that both methods of displaying grids have their own merits. During the investigation, the 'nearest integer' method was used.

Multiple Grids

Any distortion in a grid can appear very much worse when several grids with different cell sizes are displayed simultaneously. This is because the grids must 'butt' up to each other exactly and a single pixel line discrepancy would be very noticeable.

In a standard multiple grid, the cell sizes vary by a factor of three. The origins of the finer grids are specified in terms of their cell sizes from the top left corner of the coursest grid.

Using this coordinate system and the methods for displaying grids mentioned so far, the grids will possibly fail to 'butt' together. Especially if a fine grid where one cell is represented by one pixel is used.

Some work has been done to overcome these problems and grids have been 'butted' together by using a different coordinate system. In this system the finer grid origins were specified in terms of the next coursest grid cells from the next coursest grid origin.

However, this method was not robust enough to be a general case. Also it relies on non-standard result files whereas the intention was to use standard ones.

The result in that a suitable robust and general method has not yet been developed. However, this problem will be addressed in the near future.

3.3 The editor

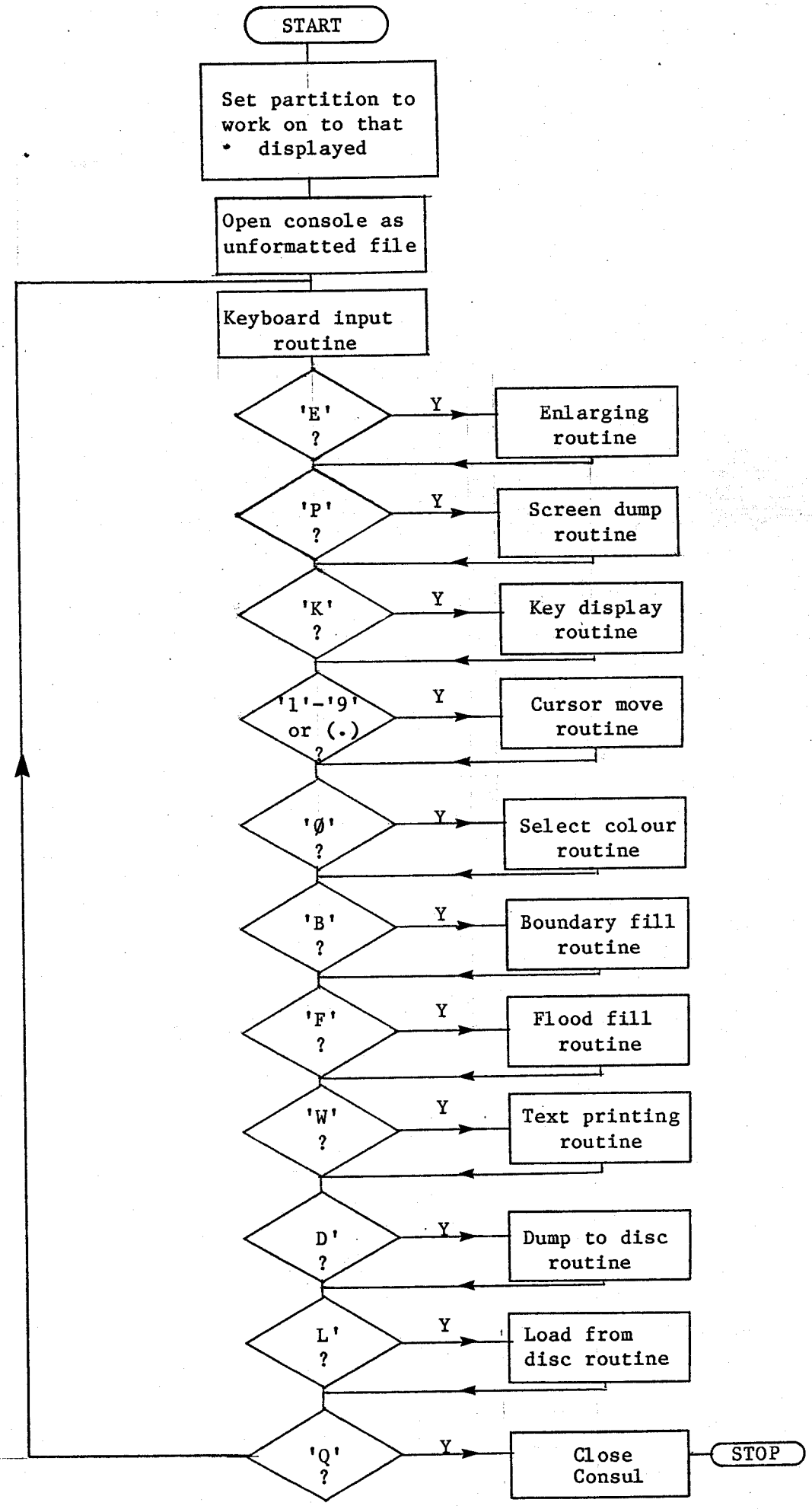
In addition to displaying images, a facility to manipulate them was also required. To satisfy this requirement a basic editing program was developed.

The editor was written in Fortran, providing an interface to the routines in the assembler library as well as other Fortran routines.

It provides a mean of altering that part of Pluto's frame buffer which is being displayed.

The user interacts with the editor by using the keyboard and single letter, keyword selection - (i.e. press 'P' for print). The cursor is manipulated by the numeric keypad on the right of the keyboard.

The editor has a modular structure. This proved valuable during the investigation since new facilities could be tried and tested simply by adding extra subroutines.


START
Set partition to work on to that displayed
Open console as unformatted file
Keyboard input routine
'E' ?
Y
Enlarging routine
'P' ?
Y
Screen dump routine
'K' ?
Y
Key display routine
'1'-'9' or (.) ?
Y
Cursor move routine
'Ø' ?
Y
Select colour routine
'B' ?
Y
Boundary fill routine
'F' ?
Y
Flood fill routine
'W' ?
Y
Text printing routine
D' ?
Y
Dump to disc routine
L' ?
Y
Load from disc routine
'Q' ?
Y
Close Consul
STOP

Enlarging routine 'E'

This routine enlarges any part of the screen by a factor of two. The area to be enlarged is shown by XORing a rectangle onto the screen.

To define the rectangle the cursor is moved to the bottom right hand corner of the area and the 'R' key is pressed, the cursor is then moved to the top left of the area and the 'L' key is pressed.

The routine checks that the rectangle does not exceed a quarter of the screen since if this were expanded it would not fit. If the dimensions are within limits the rectangle is displayed.

The cursor can then be moved to the top left corner of where the expanded box will be displayed. The 'E' key is then pressed again and the routine checks that the cursor is not too near the edge of the screen.

The rectangle is then XOR'ed again to erase it and the expanded version is displayed at the new cursor position. Each pixel is expanded into four.

Printing routine 'P'

The routine enables any part of the screen to be printed on the attached Diablo printer.

The area to be printed is defined by XORing a rectangle onto the screen in the same way that the enlarge routine does this. However, no checks are made on the rectangle dimensions.

The user is able to check that the displayed rectangle encloses the area to be printed. If it does, the rectangle is XOR'ed again to erase it and the area is dumped to the Diablo.

The screen dump routine is described more fully in the 'hard copy' section of this report.

Key display routine 'K'

This routine provides a means of displaying those colours available for line drawing, flood filling and boundary filling.

The routine displays a series of coloured rectangles along the top left corner of the screen. The colours are arranged from left to right in ascending colour code order (ie Black Ø, Green 1, Blue 2, Cyan 3, Red 4, Yellow 5, Magenta 6 and White 7).

To experiment with mixing various colours a shading routine was written for use on earlier versions of the editor.

The routine produced rectangles which consisted of vertical, pixel wide bands of any two colours. From a short distance the bands appeared to blend together into the required shade. (ie Alternate red and yellow appeared as Orange).

However, this method of obtaining extra colours is no longer available on current versions of the editor. The reasons for this are:

- The mixed colours were rather dull compared to the primaries.

- The Pluto 'pattern fill' command had to be used to 'wall paper' areas of the screen, this was slow and restrictive.

- Pluto is easily modified to an 8-bit plane model and hence 256 colours.

Select colour routine 'Ø'

This routine provides a means of changing the current and perimeter colours for line drawing, flood filling and boundary filling.

The centre of the cursor is placed over the desired colour in the key and the 'Ø' key pressed.

If the Y-coordinate of the cursor falls within the height of the key a simple algorithm is applied to the X-coordinate to determine which colour the cursor is superimposed on.

A display on the right of the screen is updated to the current colour.

Cursor move routine

The cursor is defined as a white cross hair which is superimposed onto the image without making any destructive changes.

Movement is affected by the numeric keypad on the right of the keyboard.

Key '5' acts as a toggle switch to set a trail. If the switch is 'on' a line of the current colour is displayed as the cursor moves. The trail is not

XOR'ed on to the screen so that a perimeter can be defined for boundary fills.

Key '.' acts as a 'toggle' switch to determine by how much the cursor moves. The options are 1 or 10 pixels in any direction.

Keys 1-4 and 6-9 determine the direction of movement of the cursor. Continuous movement is achieved by using the 'repeat' key.

Flood fill routine 'F'

This implements Pluto's flood fill command. The pixel at the centre of the cursor and all connected pixels of the same colour are changed to the current colour.

Boundary fill routine 'B'

This is a simple implementation of Pluto's boundary fill command.

The area enclosed by the perimeter colour is filled with the current colour.

Text printing routine 'W'

This routine enables characters to be printed on the screen in any one of four directions.

When the 'W' key is pressed the cursor is XOR'ed off the screen. The direction for printing text is selected from keys 2, 4, 6 or 8 as for cursor movement. Once the direction has been selected, any keys pressed are echoed onto the screen in that direction.

Pressing 'return' redisplays the cursor at the end of the text.

Dump to disc routine 'D'

This routine copies that part of the frame buffer being displayed to a disc file.

When the 'D' key is pressed the user is prompted for the name of a file to use. The screen is then copied to the file, unformatted.

Load from disc routine 'L'

This is the complementary routine to 'dump to disc'. The contents of the named file are loaded into the partition currently being displayed.

Figure 1

Figure 2